XX

The Space Complexity of Long-lived and One-Shot Timestamp Implementations

Maryam Helmi, University of Calgary
Lisa Higham, University of Calgary
Eduardo Pacheco, Universidad Nacional Automoma de Mexico
Philipp Woelfel, University of Calgary

This paper is concerned with the problem of implementing an unbounded timestamp object from multi-writer atomic registers, in an asynchronous distributed system of n processes with distinct identifiers where timestamps are taken from an arbitrary universe. Ellen, Fatourou and Ruppert [Ellen et al. 2008] showed that $\sqrt{n}/2 - O(1)$ registers are required for any obstruction-free implementation of long-lived timestamp systems from atomic registers (meaning processes can repeatedly get timestamps).

We improve this existing lower bound in two ways. First we establish a lower bound of $n/6 - 1$ registers for the obstruction-free long-lived timestamp problem. Previous such linear lower bounds were only known for constrained versions of the timestamp problem. This bound is asymptotically tight; Ellen, Fatourou and Ruppert [Ellen et al. 2008] constructed a wait-free algorithm that uses $n - 1$ registers. Second we show that $\sqrt{2n} - \log n - O(1)$ registers are required for any obstruction-free implementation of one-shot timestamp systems (meaning each process can get a timestamp at most once). We show that this bound is also asymptotically tight by providing a wait-free one-shot timestamp system that uses at most $\lceil 2\sqrt{n} \rceil$ registers, thus establishing a space complexity gap between one-shot and long-lived timestamp systems.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Timestamps, Solo-termination, Wait-free, Obstruction-free, Space Complexity, Shared Memory

1. INTRODUCTION

In asynchronous multiprocessor algorithms, processes have no information about the real-time order of events that are incurred by other processes. In order to solve distributed problems effectively, such as ensuring first-come-first-served fairness, or constructing synchronization primitives, it is often necessary that some reliable information about the relative order of these events can be gained.

Timestamp objects provide a means for processes to label events and then later compare those labels in order to gain information about the real-time order in which the corresponding events have occurred. Such timestamping mechanisms have been used to solve numerous problems associated with asynchrony in distributed shared memory and message passing algorithms. Examples of applications include mutual and

This work is supported by the National Sciences and Research Council of Canada Discovery Grants. E. Pacheco participated in this research while visiting the University of Calgary.
Author's addresses: M. Helmi and L. Higham and P. Woelfel, Computer Science Department, University of Calgary; E. Pacheco, Computer Science Department, University of Ottawa, Canada.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

k-exclusion algorithms [Lamport 1974; Ricart and Agrawala 1981; Fischer et al. 1989; Afek et al. 1994], consensus algorithms [Abrahamson 1988], register constructions [Haldar and Vitányi 2002; Li et al. 1996; Vitányi and Awerbuch 1986], or adaptive renaming algorithms [Attiya and Fouren 2003].

In 1978, Lamport [Lamport 1978] defined the "happens before" relation on events occurring in message passing systems to reflect the causal relationship of events. The happens before relation is a partial order, where, informally, an event e_1 happens before event e_2, if there is a causal relation that forces event e_1 to precede e_2. Lamport further devised a *logical clock* that assigns an integer value $C(e)$, called a timestamp, to each event e such that $C(e_1) < C(e_2)$ if event e_1 happens before event e_2. Lamport's logical clock system based on integers was extended to clocks based on vectors (examples include [Fidge 1988] and [Mattern 1989]) and matrices ([Wuu and Bernstein 1986] and [Sarin and Lynch 1987]).

In shared memory systems, events correspond to method invocations and responses. The happens before relation orders time intervals associated with method calls. Method call m_1 happens before method call m_2, if the response of m_1 precedes the invocation of m_2. Timestamp objects provide a mechanism to label events with timestamps from a *timestamp universe* $\mathcal{T}$ through `getTS()` (sometimes called `timestamping` or `label`) method calls. If $\mathcal{T}$ is finite, then the timestamp object is said to be *bounded*, otherwise it is *unbounded*.

Often, $\mathcal{T}$ is a partially ordered set, and all timestamps returned by `getTS()` method calls during an execution preserve the happens before relation of these method calls. Such timestamp objects are called *static*. Non-static timestamp objects can take the current system state into account when comparing the order of two timestamps. Thus, different executions can lead to different partial orders of the set $\mathcal{T}$. Sometimes, in particular when $\mathcal{T}$ is bounded, the happens before relation is only preserved for a subset of *valid* timestamps in $\mathcal{T}$, e.g., the set of the last timestamps obtained by each process. In this case, timestamp objects often provide a `scan` method that returns an ordered list of all valid timestamps. The literature contains several examples of constructions of bounded and unbounded timestamp objects [Lamport 1974; Gawlick et al. 1992; Israeli and Pinhasov 1992; Israeli and Li 1993; Dolev and Shavit 1997; Dwork and Waarts 1999; Haldar and Vitányi 2002; Attiya and Fouren 2003; Guerraoui and Ruppert 2007; Ellen et al. 2008].

Ellen, Fatourou, and Rupert [Ellen et al. 2008] studied the number of atomic registers needed to implement timestamp objects. In order to prove strong lower bounds, the authors considered a very weak definition of an unbounded non-static timestamp object, that, in addition to `getTS()` provides a method $\texttt{compare}(t_1, t_2)$ for two timestamps $t_1, t_2 \in \mathcal{T}$. The only requirement is that if a `getTS()` method g_1 that returns t_1 happens before another `getTS()` method g_2 that returns t_2 then any later $\texttt{compare}(t_1, t_2)$ must return true and any later $\texttt{compare}(t_2, t_1)$ must return false.

As their main result, Ellen et al. showed that any implementation that satisfies non-deterministic solo-termination (a progress condition weaker than wait-freedom or obstruction-freedom, and that is defined in Section 2) requires at least $\frac{1}{2}\sqrt{n-1}$ registers, where n is the number of processes in the system. Despite the weak requirements, the best known algorithm (also in [Ellen et al. 2008]) needs $n-1$ registers, leaving a large gap between the best known lower and upper bounds. However, for two stronger versions of the problem, Ellen et al. obtain tight lower bounds, showing that n registers are necessary, first, for static algorithms, where $\mathcal{T}$ is *nowhere dense* (i.e., any two elements $x, y \in \mathcal{T}$ satisfy $|\{z \in \mathcal{T} | x < z < y\}| < \infty$), and second, for anonymous algorithms.

Our Contributions. We distinguish between *one-shot* timestamp objects, where each process is allowed to call `getTS()` at most once, and *long-lived* ones, where each process can call `getTS()` arbitrarily many times. (In either case, the number of `compare` methods calls is not restricted.) We first improve the $\Omega(\sqrt{n})$ lower bound of [Ellen et al. 2008] for long-lived timestamp objects to an asymptotically tight one:

THEOREM 1.1. *Any long-lived unbounded timestamp object that satisfies non-deterministic solo-termination uses at least $n/6-1$ registers.*

Therefore, even under very weak assumptions, at least linear register space is necessary. Since it is not possible to implement general timestamp objects using sublinear space, it makes sense to look at restricted solutions.

Several methods have solutions that are simpler than the general case, if each process is allowed to execute it only once. Examples are renaming and mutual exclusion algorithms, splitter or snapshot objects, or agreement problems. Other problems, such as consensus or non-resettable test and set objects are inherently "one-time". It is conceivable that if an implementation of such an algorithm uses timestamp objects, then in the "one-shot" version of that algorithm each process needs to obtain a timestamp only once. Therefore, we study the space complexity of one-shot timestamp objects:

THEOREM 1.2. *Any one-shot unbounded timestamp object that satisfies non-deterministic solo-termination uses at least $\sqrt{2n}-\log n-O(1)$ registers.*

This one-shot lower bound is a factor of approximately $2\sqrt{2}$ larger than the previous best known lower bound for the long-lived case [Ellen et al. 2008], and holds for historyless objects as well as registers as explained later.

THEOREM 1.3. *There is a wait-free implementation of one-shot timestamp objects that uses $2\lceil\sqrt{n}\rceil$ registers.*

Our lower bound proofs are based on covering arguments (as introduced by Burns and Lynch [Burns and Lynch 1993]), where one constructs an execution in which processes are poised to write to some registers (the processes are said to *cover* these registers). We rely on a lemma by Ellen, Fatourou and Ruppert [Ellen et al. 2008] that shows how in a situation where some processes cover a set R of registers, other processes can be forced to write outside of R. In order to obtain our improved lower bound for the long-lived case, we look at very long executions in which "similar" coverings are obtained over and over again. Our lower bound proof for the one-shot case is inspired by a geometric interpretation of the covering structure of configurations. The one-shot timestamps upper bound exploits the structure exposed by the lower bound.

2. PRELIMINARIES

We consider an asynchronous shared memory system with a set $\mathcal{P}=\{p_1,\dots,p_n\}$ of n processes and a set $\mathcal{R}=\{r_1,\dots,r_m\}$ of m registers that support atomic read and write operations. Processes can only communicate via those operations on shared registers. We assume that processes can make arbitrary non-deterministic decisions, but we require that the result of any execution is correct, meaning that the responses from method calls match the specification of timestamp objects.

A *configuration* C is a tuple $(s_1,\dots,s_n,v_1,\dots,v_m)$, denoting that process p_i, $1\le i\le n$, is in state s_i, and register r_j, $1\le j\le m$, has value v_j. Configurations will be denoted by capital letters, and the initial configuration is denoted C_0.

An implementation of a method satisfies *non-deterministic solo-termination*, if for any configuration C and any process p_i, $1\le i\le n$, there is an execution in which no process other than p_i takes any steps, and p_i finishes its method call within a fi-

nite number of steps [Fich et al. 1998]. Hence, a process is guaranteed to finish its method call with positive probability, whenever there is no interference from other processes. For deterministic algorithms, non-deterministic solo-termination is the same as obstruction-freedom and weaker than wait-freedom. Both our lower bound results hold for timestamp objects that satisfy this progress condition, our algorithm, however, satisfies the stronger wait-free progress property.

A *schedule* σ is a (possibly infinite) sequence of process indices. An *execution* $(C;\sigma)$ is a sequence of steps beginning in configuration C and moving through successive configurations one at a time. At each step, the next process p_i indicated in the schedule σ, takes the next step in its program. Since our computation model is non-deterministic, we fix the non-deterministic decision made by p_i in our lower bound proofs. We use an arbitrary (but fixed) one that guarantees that p_i terminates within a bounded number of steps if it executes alone. If σ is a finite schedule, the final configuration of the execution $(C;\sigma)$ is denoted $\sigma(C)$. If σ and π are finite schedules then $\sigma\pi$ denotes the concatenation of σ and π. Let P be a set of processes, and σ a schedule. If only indices of processes in P appear in σ, then σ is a P*-only* schedule and any execution $(C;\sigma)$ is a P*-only* execution. If $|P| = 1$, a P-only schedule σ is a *solo* schedule and any execution $(C;\sigma)$ is a *solo* execution.

A configuration, C, is *reachable* if there exists a finite schedule, σ, such that $\sigma(C_0) = C$.

Any execution $(C;\sigma)$ defines a partial *happens before* order "$\rightarrow$" on the method calls that occur during $(C;\sigma)$. A method call m_1 happens before m_2, denoted $m_1 \rightarrow m_2$, if the response of m_1 occurs before the invocation of m_2.

An unbounded timestamp object supports two methods, `getTS()` and `compare()`. The first one outputs a *timestamp* without receiving any input; the `compare` method receives any two timestamps as inputs, and returns true or false. If two `getTS()` instances g_1 and g_2 return t_1 and t_2, respectively, and $g_1 \rightarrow g_2$, then `compare`(t_1, t_2) returns true and `compare`(t_2,t_1) returns false.

A timestamp object is *long-lived*, if each process is allowed to invoke `getTS()` multiple times; it is *one-shot* when each process is allowed to invoke `getTS()` only once.

Our lower bounds are based on covering arguments. We will construct executions, at the end of which processes are poised to write, i.e., they *cover* several registers. If other process are scheduled after this and if they write only to the same set of registers, their trace can be eliminated. More precisely, we say process p_i *covers* register r_j in a configuration C, if there is a non-deterministic decision such that the one step execution $\big(C;(i)\big)$ is a write to register r_j. A set of processes P covers a set of registers R if for every register $r \in R$ there is a process $p \in P$ such that p covers r.

For a process set P, let π_P denote an arbitrary (but fixed) permutation of P (for example the one that orders processes by their ID). If the process set P covers the register set R in configuration C, the information held in the registers in R can be overwritten by letting all processes in P execute exactly one step. Such an execution by the processes in P is called a *block-write*. More precisely, a *block-write* by P to R is an execution $(C;\pi_P)$.

Two configurations $C_1 = (s_1,\ldots,s_n,r_1,\ldots,r_m)$ and $C_2 = (s'_1,\ldots,s'_n,r'_1,\ldots,r'_m)$ are *indistinguishable* to process p_i if $s_i = s'_i$ and $r_j = r'_j$ for $1 \le j \le n$. If S is a set of processes, and for every process $p \in S$, C_1 and C_2 are indistinguishable to p, then for any S-only schedule σ, $\sigma(C_1)$ and $\sigma(C_2)$ are indistinguishable to p.

Our first lower lower bound relies on a lemma which is based on the following observation. Suppose in configuration C there are three disjoint sets of processes B_0, B_1, B_2, each covering a set R of registers, and q_0 and q_1 are processes not in $B_0 \cup B_1 \cup B_2$. Let σ_i, $i \in \{0,1\}$, denote an arbitrarily long $\{q_i\}$-only schedule. If, for $i \in \{0,1\}$, in the

execution $(C; \pi_{B_i}\sigma_i)$, q_i does not write outside R, then the configurations $\pi_{B_i}\sigma_i(C)$ and $\pi_{B_{i-1}}\sigma_{i-1}\pi_{B_i}\sigma_i(C)$ are indistinguishable to q_i. Furthermore, after a subsequent third block write by B_2 all trace left inside of R can also be obliterated. Thus, the configurations $C_0 = \pi_{B_0}\sigma_0\pi_{B_1}\sigma_1\pi_{B_2}(C)$ and $C_1 = \pi_{B_1}\sigma_1\pi_{B_0}\sigma_0\pi_{B_2}(C)$ are indistinguishable to all processes, unless at least one of either q_0 or q_1 writes outside R. If, however, the solo executions by q_0 and q_1 both contain complete `getTS()` calls, then one happens after the other and so processes have to be able to distinguish between C_0 and C_1. Hence, either q_0 or q_1 writes outside R in both of the executions $(C; \pi_{B_0}\sigma_0\pi_{B_1}\sigma_1)$ and $(C; \pi_{B_1}\sigma_1\pi_{B_0}\sigma_0)$.

The same idea works if we replace q_0 and q_1 with disjoint sets of processes, as was done in the original version of this lemma due to Ellen, Fatourou, and Rupert [Ellen et al. 2008]. We state a simplified form here that suffices for our results and uses the form and notation of this paper.

LEMMA 2.1 ([ELLEN ET AL. 2008]). *Consider any timestamp implementation from registers that satisfies non-deterministic solo-termination and let C be a reachable configuration. Let B_0, B_1, B_2, U_0, U_1 be disjoint sets of processes, where in C each of B_0, B_1, and B_2 cover a set R of registers. Then there exists $i \in \{0, 1\}$ such that every U_i-only execution starting from $C_i = \pi_{B_i}(C)$ that contains a complete* `getTS()` *method writes to some register not in R.*

Our second lower bound relies on a stronger lemma that is proved by inductively applying Lemma 2.1.

3. A SPACE LOWER BOUND FOR LONG-LIVED TIMESTAMPS

We assume that a timestamp object is used in an algorithm where each process calls `getTS()` infinitely many times. Actually, the number of `getTS()` calls can be bounded (by a function growing exponentially in n), but for convenience we pass on computing this bound. Ellen et al. used their lemma in order to inductively construct executions at the end of which k registers are covered by $\Omega(\sqrt{n} - k)$ processes, where k is bounded by $O(\sqrt{n})$. The lemma is used in the inductive step to show that in some execution following a block-write, many of the non-covering processes can be forced to write outside the set of covered registers. By the pigeon hole principle, one additional (previously not covered) register can then be covered with many processes. With this idea, however, the number of processes covering one register is reduced by one in each inductive step, and thus it is not hard to see that the technique cannot lead to a lower bound beyond $\Omega(\sqrt{n})$.

In our proof, rather than requiring that many processes cover the same register, we limit the number of processes covering the same register to three. In particular, we define a $(3, k)$-configuration to be one where k processes are covering registers, but no register is covered by more than three of them. Using an argument reminiscent of that used by Burns and Lynch [Burns and Lynch 1993], we show that if there is an execution that leads to some $(3, k)$-configuration, we can find a (much longer) execution, during which at least two $(3, k)$-configurations C_1 and C_2 are encountered that are similar in the sense that in both configurations each register is covered by the same number of processes. In addition, the execution $(C_1; \sigma)$ that leads from C_1 to C_2 starts with three block-writes to the registers that are covered by three processes, each. We then apply Lemma 2.1 to see that we can insert a p-only schedule for some unused process p into the schedule σ after one of the block-writes to get the new schedule σ', such that at the end of the execution $(C_1; \sigma')$ process p is poised to write outside of the registers that are 3-covered in C_1. Since the other two block-writes overwrite p's trace in $(C_1; \sigma')$, no process other than p can distinguish between $\sigma'(C_1)$ and $\sigma(C_1) = C_2$. It follows that in $\sigma'(C_1)$ process p covers a register that is covered by at most 2 other processes. Hence, we have obtained a $(3, k + 1)$-configuration. We can do this for $k \leq n/2$,

so in the end we obtain a $(3, \lfloor n/2 \rfloor)$-configuration. Clearly, this means that the number of registers is at least $\lfloor n/6 \rfloor$.

The signature of a configuration C, denoted $\mathrm{sig}(C)$, is a tuple $(c_1, c_2, \ldots, c_m)$ where every c_i is the number of processes covering the i-th register in C. The set of registers whose corresponding entry in $\mathrm{sig}(C)$ is equal to 3 is denoted $\mathcal{R}_3(C)$. (In terms of signatures, a configuration C is a $(3, k)$-configuration if $\mathrm{sig}(C) = (c_1, c_2, \ldots, c_m)$ satisfies $\sum_{i=1}^{m} c_i = k$ and $c_i \leq 3$ for every $1 \leq i \leq m$.) Notice that in any $(3, k)$-configuration there are at least $\lceil k/3 \rceil$ registers covered. Configuration C is *quiescent* if in C no process has started but not finished executing a `getTS()` or `compare()` call.

LEMMA 3.1. *Let P be an arbitrary set of processes. Suppose for every reachable quiescent configuration D there exists a P-only schedule σ such that $\sigma(D)$ is a $(3, k)$-configuration. Then for any quiescent configuration D, there are two $(3, k)$-configurations C_0 and C_1, and P-only schedules γ_0, γ_1, and η such that:*

(a). $\gamma_0(D) = C_0$,
(b). $\gamma_1(C_0) = C_1$,
(c). $\mathrm{sig}(C_0) = \mathrm{sig}(C_1)$, *and*
(d). $\gamma_1 = \pi_{B_0}\pi_{B_1}\pi_{B_2}\eta$, *where B_0, B_1 and B_2 are disjoint sets of processes each covering $\mathcal{R}_3(C_0)$.*

PROOF. We inductively define an infinite sequence of schedules $\lambda_0, \delta_0, \lambda_1, \delta_1, \ldots, \lambda_i, \delta_i, \ldots$ and reachable $(3, k)$-configurations $E_0, E_1, E_2, \ldots$, where $E_{i+1} = \lambda_i \delta_i(E_i)$, as follows. E_0 is the $(3, k)$-configuration $\sigma(D)$ guaranteed by the hypothesis of the lemma. Let $B_{0,i}, B_{1,i}$ and $B_{2,i}$ be disjoint sets of processes each covering $\mathcal{R}_3(E_i)$. Execution $(E_i; \pi_{B_{0,i}}\pi_{B_{1,i}}\pi_{B_{2,i}})$ consists of three consecutive block-writes to $\mathcal{R}_3(E_i)$ by the processes in $B_{0,i}$, $B_{1,i}$, and $B_{2,i}$, respectively. Schedule λ_i is the concatenation of the sequence of permutations $\pi_{B_{0,i}}\pi_{B_{1,i}}\pi_{B_{2,i}}$ and some P-only schedule α_i in which every process in P with a pending operation, finishes that pending operation. Thus, configuration $\lambda_i(E_i) = \pi_{B_{0,i}}\pi_{B_{1,i}}\pi_{B_{2,i}}\alpha_i(E_i)$ is quiescent. So by the hypothesis there exists a schedule δ_i such that $E_{i+1} = \lambda_i\delta_i(E_i)$ is again a $(3, k)$-configuration.

Since the set of signatures is finite, there are two indices $j < k$, such that $\mathrm{sig}(E_j) = \mathrm{sig}(E_k)$. Fix two such indices j and k. Let $\gamma_0 = \sigma\lambda_0\delta_0\lambda_1\delta_1\lambda_2\delta_2 \ldots \lambda_{j-1}\delta_{j-1}$ and $\gamma_1 = \lambda_j\delta$ where $\delta = \delta_j\lambda_{j+1}\delta_{j+1} \ldots \lambda_{k-1}\delta_{k-1}$. Furthermore, let $C_0 = \gamma_0(D)$ and $C_1 = \gamma_1(C_0)$. By definition, the configurations C_0 and C_1 satisfy (a) and (b). Moreover, by construction $C_0 = E_j$ and $C_1 = E_k$ and since $\mathrm{sig}(E_j) = \mathrm{sig}(E_k)$, (c) is satisfied. Finally, let $\eta = \alpha_j\delta$. Then, $\gamma_1 = \pi_{B_{0,j}}\pi_{B_{1,j}}\pi_{B_{2,j}}\eta$, where $B_{0,j}, B_{1,j}, B_{2,j}$ are disjoint sets of processes each covering $\mathcal{R}_3(E_j) = \mathcal{R}_3(C_0)$. This proves (d). $\square$

Let $\mathcal{P}_k$ denote the set $\{p_1, \ldots, p_k\}$ and P_0 denote the emptyset of processes.

LEMMA 3.2. *For every $0 \leq k \leq \lfloor n/2 \rfloor$ and for every reachable quiescent configuration D, there exists a $\mathcal{P}_{2k}$-only schedule σ_k such that $\sigma_k(D)$ is a $(3, k)$-configuration.*

PROOF. The proof is by induction on k. For $k = 0$ the claim is immediate by choosing σ_0 to be the empty schedule.

Let $k \geq 1$, and let D be an arbitrary reachable quiescent configuration. By the induction hypothesis, for every reachable quiescent configuration C, there exists a $\mathcal{P}_{2k-2}$-only schedule σ_{k-1}, such that $\sigma_{k-1}(C)$ is a $(3, k-1)$-configuration. Hence, by Lemma 3.1 with $P = \mathcal{P}_{2k-2}$ there are two reachable configurations C_0 and C_1, and $\mathcal{P}_{2k-2}$-only schedules γ_0, γ_1, and η, such that $\gamma_0(D) = C_0$, $\gamma_1(C_0) = C_1$, $\mathrm{sig}(C_0) = \mathrm{sig}(C_1)$, and $\gamma_1 = \pi_{B_0}\pi_{B_1}\pi_{B_2}\eta$, where B_0, B_1 and B_2 are disjoint sets of processes, each covering $\mathcal{R}_3(C_0)$.

Consider the two processes p_{2k-1} and p_{2k}. For $i \in \{0,1\}$, let α_i be a $\{p_{2k-i}\}$-only schedule such that in execution $(\pi_{B_i}(C_0);\alpha_i)$, p_{2k-i} performs a complete getTS() instance. According to Lemma 2.1, there exists $i \in \{0,1\}$, such that p_{2k-i} writes to some register not in $\mathcal{R}_3(C_0)$ during the execution $(\pi_{B_i}(C_0);\alpha_i)$. (Note that whether $i=0$ or $i=1$ depends on C_0.) Let r be the first register not in $\mathcal{R}_3(C_0)$ to which p_{2k-i} writes to in $(\pi_{B_i}(C_0);\alpha_i)$. Since $\mathrm{sig}(C_0)=\mathrm{sig}(C_1)$, we have $r \notin \mathcal{R}_3(C_1)$, and thus r is covered by at most two processes in C_0 as well as in C_1.

Let λ be the shortest prefix of α_i such that p_{2k-i} is about to write to r in $\pi_{B_i}\lambda(C_0)$. Since p_{2k-i} does not participate in schedule $\pi_{B_{1-i}}\pi_{B_2}\eta$, it is also covering r in the configuration $\pi_{B_i}\lambda\pi_{B_{1-i}}\pi_{B_2}\eta(C_0)$. Configurations $\pi_{B_i}\pi_{B_{1-i}}\pi_{B_2}(C_0)$ and $\pi_{B_{1-i}}\pi_{B_i}\pi_{B_2}(C_0)$ are indistinguishable to all processes; therefore, $\pi_{B_i}\pi_{B_{1-i}}\pi_{B_2}\eta(C_0)=C_1$. Moreover, since $C_1=\pi_{B_0}\pi_{B_1}\pi_{B_2}\eta(C_0)$ is indistinguishable from $\pi_{B_i}\lambda\pi_{B_{1-i}}\pi_{B_2}\eta(C_0)$ to every process except p_{2k-i}, all processes other than p_{2k-i} cover the same registers in C_1 as in $\pi_{B_i}\lambda\pi_{B_{1-i}}\pi_{B_2}\eta(C_0)$. Since p_{2k-i} covers r in this configuration, and r is covered by at most 2 other processes, $\pi_{B_i}\lambda\pi_{B_{1-i}}\pi_{B_j}\eta(C_0)$ is a $(3,k)$-configuration. $\square$

Lemma 3.2 shows that in any long-lived unbounded timestamp implementation that satisfies non-deterministic solo-termination there exists a reachable $(3,\lfloor n/2\rfloor)$-configuration. Clearly, at least $\lfloor n/6\rfloor > n/6-1$ registers are covered in this configuration. This proves Theorem 1.1.

4. A SPACE LOWER BOUND FOR ONE-SHOT TIMESTAMPS

It seems natural to imagine that n registers would be required to construct a timestamp system for n processes. But this is not the case for some restricted versions of the problem. For example, if the timestamps are not required to come from a nowhere dense set, then, as shown by Ellen, Fatourou and Ruppert [Ellen et al. 2008], $n-1$ registers suffice. We show that another instance is when each process is restricted to at most one call to the getTS() method. In this case $\Theta(\sqrt{n})$ registers are necessary and sufficient. This section contains the space lower bound. Section 6 contains the algorithm that shows that this lower bound is asymptotically tight.

Our lower bound proof relies on Lemma 4.1, the proof of which uses Lemma 2.1 inductively. Given four disjoint sets of processes B_1, B_2, B_3, U such that processes in B_1, B_2, B_3 cover a set of registers R, then, according to Lemma 2.1, for any partition of U into V_1 and V_2, either all the processes in V_1 or all the processes in V_2 can be made to cover some register outside of R. By choosing V_1 and V_2 to have sizes differing by at most one, Lemma 2.1 can be used to ensure that essentially half of the processes in $V_1 \cup V_2$ must write outside of R.

We strengthen this idea by using Lemma 2.1 inductively to construct an execution such that all but one of the processes in U that have not initiated any operation can be made to cover some register outside of the set of registers R. Let $\mathrm{participants}(\sigma)$ denote the set of the processes taking steps in schedule σ. A process is *idle* in configuration C if it is in its initial state in C; the set of all such processes is denoted $\mathrm{idle}(C)$.

LEMMA 4.1.
Let C be a reachable configuration of a one-shot timestamp implementation from registers that satisfies non-deterministic solo-termination. Let B_0, B_1, B_2, U be disjoint sets of processes where in C each of B_0, B_1 and B_2 cover a set R of registers and $U \subseteq \mathrm{idle}(C)$, with $|U| \geq 2$. Then there is a schedule $\beta\sigma\beta'\sigma'$ satisfying:

(a) $\{\beta,\beta'\} = \{\pi_{B_0},\pi_{B_1}\}$;
(b) In configuration $\beta\sigma\beta'\sigma'(C)$ all processes in $\mathrm{participants}(\sigma)$ and $\mathrm{participants}(\sigma')$ cover a register outside of R;
(c) $\mathrm{participants}(\sigma) \cup \mathrm{participants}(\sigma') \subsetneq U$.

(d) $|\text{participants}(\sigma)| + |\text{participants}(\sigma')| = |U| - 1$*;*
(e) $|\text{participants}(\sigma)| \geq \lfloor |U|/2 \rfloor \geq |\text{participants}(\sigma')|$*;*
(f) σ *and* σ' *are concatenations of solo schedules by distinct processes in* U*.*

PROOF. Let $U = \{p_0, \ldots, p_m\}$, where $m \geq 1$ (because $|U| \geq 2$). For each $1 \leq k \leq m$, we first inductively construct schedules δ_0^k and δ_1^k such that

— $\text{participants}(\delta_0^k)$ and $\text{participants}(\delta_1^k)$ form a partition of $\{p_0, \ldots p_k\}$;
— for $i \in \{0, 1\}$, in execution $(C; \pi_{B_i}\delta_i^k)$:
  — each process in $\text{participants}(\delta_i^k)$ initiates exactly one instance of `getTS()`;
  — exactly one `getTS()` method completes, and this `getTS()` is by the last process in δ_i^k;
  — no process except possibly the last that occurs in δ_i^k writes outside of R;
— for $i \in \{0, 1\}$, in configuration $\pi_{B_i}\delta_i^k(C)$ every process in $\text{participants}(\delta_i^k)$ except possibly the last that occurs in δ_i^k cover a register outside of R.

For $i \in \{0, 1\}$, let δ_i^1 be a p_i-only schedule in which process p_i performs a complete `getTS()` instance in the execution $(C; \pi_{B_i}\delta_i^1)$. Such a schedule δ_i^1 exists because p_0 and p_1 are in $\text{idle}(C)$. This immediately satisfies the base case, $k = 1$.

For $i \in \{0, 1\}$, suppose that δ_i^k are constructed as required, and let q_i denote the last process in δ_i^k. Since execution $(C; \pi_{B_i}\delta_i^k)$ contains a complete `getTS()` by q_i, and no process in δ_i^k before q_i writes outside of R in $(C; \pi_{B_i}\delta_i^k)$, Lemma 2.1 implies that either q_0 in execution $(\pi_{B_0}(C); \delta_0^k)$ or q_1 in execution $(\pi_{B_1}(C); \delta_1^k)$ must write outside of R. Choose such a $j \in \{0, 1\}$ such that process q_j does write outside of R in $(\pi_{B_j}(C); \delta_j^k)$. First truncate the schedule δ_j^k , to, say, α_j^k, by deleting a suffix of the solo schedule of q_j so that, instead of completing its `getTS()` method, q_j is paused at the earliest point such that at the end of the execution $(\pi_{B_j}(C); \alpha_j^k)$, q_j covers a register outside of R. Now append to α_j^k, a p_{k+1}-only schedule σ_{k+1} so that the execution $(\pi_{B_j}(C); \alpha_j^k\sigma_{k+1})$ contains a complete `getTS()` method by p_{k+1}. Define δ_j^{k+1} to be $\alpha_j^k\sigma_{k+1}$ and δ_{1-j}^{k+1} to be δ_{1-j}^k. The claimed construction now holds for $k + 1$.

Therefore, we can construct two schedules, δ_i^m for $i \in \{0, 1\}$, that together contain all the processes of U and where each is a concatenation of distinct solo-executions. Furthermore, each of the executions $(C; \pi_{B_i}\delta_i^m)$ contains exactly one complete `getTS()` by the last process in the schedule δ_i^m, and no other process writes outside of R. Therefore, applying Lemma 2.1 one more time, for a $j \in \{0, 1\}$, in the execution $(C; \pi_{B_j}\delta_j^m)$, the last process in δ_j^m must write outside of R. Let σ_j be the schedule δ_j^m truncated to the first point such that at the end of execution $(C; \pi_{B_j}\sigma_j)$ this last process covers a register outside of R. Let σ_{1-j} be the schedule δ_{1-j}^m truncated to remove the entire schedule of its last process.

Now relabel the members of $\{\pi_{B_0}\sigma_0, \pi_{B_1}\sigma_1\}$ to have distinct names in $\{\beta\sigma, \beta'\sigma'\}$ in such a way that the two schedules σ_0 and σ_1 are renamed with distinct names in $\{\sigma, \sigma'\}$ and satisfy $|\text{participants}(\sigma)| \geq |\text{participants}(\sigma')|$. By construction, $\text{participants}(\sigma_0)$ and $\text{participants}(\sigma_1)$ do not intersect; each is a subset of U; and together they contain all but 1 of the members of U. Also, by construction, each of σ_0 and σ_1 are concatenations of solo executions. When combined with the relabeling, this establishes (a), (c), (d), (e) and (f).

Since $\text{participants}(\sigma_0)$ and $\text{participants}(\sigma_1)$ are disjoint sets, and since no process writes outside of R in the execution $(C; \pi_{B_i}\sigma_i)$ for $i \in \{0, 1\}$, and since each block write obliterates all writes to R, configurations $\pi_{B_i}(C)$ and $\pi_{B_{1-i}}\sigma_{1-i}\pi_{B_i}(C)$ are indistinguishable to $\text{participants}(\sigma_i)$. So each process in $\text{participants}(\sigma_i)$ covers the same register in $\pi_{B_i}\sigma_i(C)$ as it does in $\pi_{B_{1-i}}\sigma_{1-i}\pi_{B_i}\sigma_i(C)$ and as it does in $\pi_{B_i}\sigma_i\pi_{B_{1-i}}\sigma_{1-i}(C)$.

Consequently, in both $\pi_{B_0}\sigma_0\pi_{B_1}\sigma_1(C)$ and $\pi_{B_1}\sigma_1\pi_{B_0}\sigma_0(C)$ each of the $m-1$ processes in $\mathrm{participants}(\sigma_0) \cup \mathrm{participants}(\sigma_1)$ covers a register not in R. This, combined with the relabeling, establishes (b). $\square$

Lemma 4.1 is the principle tool for our space lower bound for one-shot timestamps. To describe the structure of the proof we use the following definitions. Let $m = \lfloor\sqrt{2n}\rfloor$. Assume that the set of all registers, denoted $\mathcal{R}$, has size at most m since otherwise we are done. Define the *ordered-signature of a configuration* C, denoted $\mathrm{ordSig}(C)$, to be the m-tuple $(s_1, s_2, \ldots, s_m)$ where $s_i \geq s_{i+1}$, and there is a permutation α of $\{1, \ldots, m\}$ such that for $1 \leq i \leq m$, s_i processes are covering the α_i-th register. (The ordered-signature of a configuration is just its signature with the entries of the m-tuple reordered so that they are non-increasing. If only $k < m$ registers exist then $s_{k+1} = s_{k+2} = \ldots = s_m = 0$.) A configuration C with $\mathrm{ordSig}(C) = (s_1, \ldots, s_m)$, is ℓ-*constrained* if $s_c \leq \ell - c$ for every $1 \leq c \leq \ell$. A configuration C is (j,k)-*full* if there is a set R of registers such that $|R| = j$ and in C each register in R is covered by at least k processes. If C is (j,k)-full, $\mathcal{R}_{j,k}(C)$ denotes a set of such registers, otherwise $\mathcal{R}_{j,k}(C)$ is undefined.

If C is (j,k)-full where $k \geq 3$, and there are $u \geq 2$ processes that are idle in C, then Lemma 4.1 can be applied with B_0, B_1, B_2 any 3 disjoint sets each covering $\mathcal{R}_{j,k}(C)$, so that for any $1 \leq v \leq u-1$, v processes can be made to cover registers outside of $\mathcal{R}_{j,k}(C)$ using at most 2 block writes to $\mathcal{R}_{j,k}(C)$.

We use this idea repeatedly to construct an execution that visits a sequence of configurations, say $C_1, ..., C_{\text{last}}$ such that the set of registers covered in C_{i+1} is a superset of the set covered in C_i until eventually a configuration C_{last} is reached in which at least $m - \log n$ registers are covered.

Intuition for our construction is aided by a geometric representation of configurations. Configuration C with $\mathrm{ordSig}(C) = (s_1, s_2, \ldots, s_m)$ is represented on a grid of cells where, in each column c, $1 \leq c \leq m$, the lowest s_c cells are shaded. Thus each register corresponds to a column in the grid, but this correspondence can change for different configurations. With this interpretation, each shaded cell in column c represents a process covering the register corresponding to c. If the configuration is l-constrained, the shading in each column remains below the stepped diagonal that starts at height $l-1$ in the grid. The configuration is (j,k)-full if in column j (and hence in all columns 1 through j) the height of the shaded cells is at least k.

An overview of the construction is as follows. We first achieve an m-constrained $(j, m-j)$-full configuration for some $j \geq 1$ as shown in Figure 1.

Given some ℓ-constrained $(j, \ell-j)$-full configuration, (such as shown in Figure 1 with $m = \ell$) and provided $\ell - j$ is at least 3, we can apply Lemma 4.1 using 3 disjoint sets of processes each occupying cells in columns 1 through j for the sets B_0, B_1 and B_2. Then, one at a time, idle processes can be made to occupy cells in columns $j+1$ through m. We will maintain the invariant that the number of idle processes is always greater than the number of unshaded cells that are under the stepped diagonal and in columns $j+1$ through m. Because of this invariant, we can be sure to reach a configuration C' where, for the first time, (when the columns $j+1$ through m are rearranged in order of non-increasing number of occupants) some column $j' \geq j+1$ gets $\ell - j'$ occupants. During this execution the block writes reduced the height of the shaded cells in columns 1 through j by one or two. If only one block write happened during this execution, or if $j' \geq j+2$, C' is again an ℓ-constrained $(j', \ell-j')$-full configuration (Case 1 of Figure 2).

The only other case is when both block writes were used to achieve C' and $j' = j+1$ (Case 2 of Figure 2). Then C' is an $(\ell-1)$-constrained $(j', \ell-1-j')$-full configuration. In this case, however, at least half of the idle processes have moved to occupy cells in columns $j+1$ through m. So this reduction by one in the stepped boundary of the

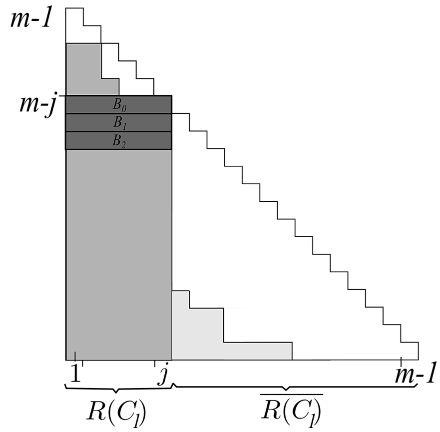

Fig. 1. Configuration C_1 must have a column j that reaches to the diagonal. Hence there are j registers each covered with $m - j$ processes.

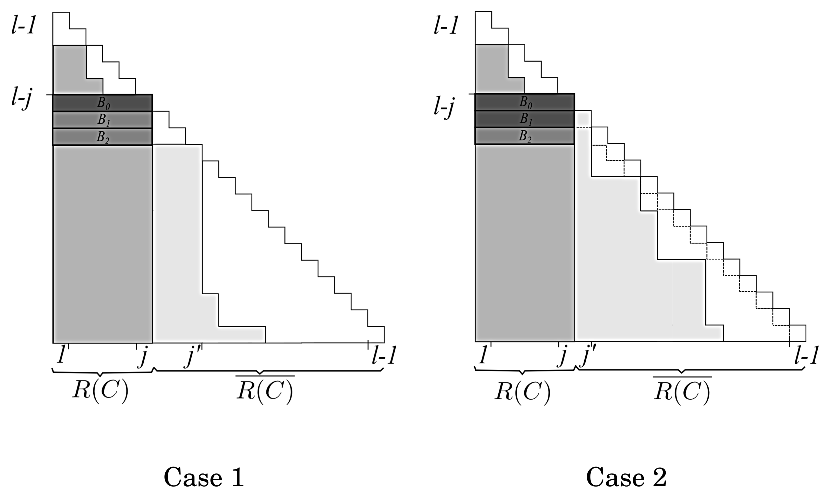

Fig. 2. After the block-write, processes are run until some new column j' reaches the diagonal and thus has height $\ell - j'$. Case 1: columns 1 through j still have height at least $\ell - j'$. Case 2: the diagonal is reached at column $j + 1$ after two block writes. This can only happen if at least half of the unshaded space in columns $j + 1$ through m became shaded.

grid can only happen $\log n$ times. Thus, each repetition of this construction creates a $(m-s)$-constrained $(k, m-k-s)$-full configuration where $s \in O(\log n)$. The construction can be repeated until either there are fewer than 2 idle processes or $m - k - s < 3$. In both cases at least $m - s = \sqrt{2n} - O(\log n)$ registers are covered.

The rest of this section contains the details of this construction, which provides the proof of Theorem 1.2. We assume that $n \geq 3$ since otherwise the theorem is trivially correct. For configuration C, and a set of registers $R \subseteq \mathcal{R}$, $\mathrm{poised}(C, R)$ denotes the processes that are covering some register in R. For any set of registers R, $\overline{R}$ denotes the set $\mathcal{R} \setminus R$.

The construction is inductive, starting with the initial configuration C_0. Initialize $j_0 = 0$, $\ell_0 = m$ and $R_0 = \emptyset$.

In C_0, no register is covered. Set $B_0 = B_1 = B_2 = \emptyset$, let U be the set of all n processes and apply Lemma 4.1. Because π_{B_i} is the empty schedule for $i \in \{0, 1, 2\}$, the schedule produced is $\sigma\sigma'$ and $n - 1$ processes cover some register in configuration $\sigma\sigma'(C_0)$. Let $\mathrm{ordSig}(\sigma\sigma'(C_0)) = (s_1, s_2, \dots, s_m)$. If $s_m \geq 1$, we are done since then m registers are covered. Therefore, assume $s_m = 0$, and suppose $s_c \leq m - c - 1$ for each $c \leq m - 1$. Then, $\sum_{c=1}^{m} s_c \leq \sum_{c=1}^{m-1}(m - c - 1) + s_m = (m - 1)(m - 2)/2 + 0 < n - 1$, which is impossible. Hence, there is at least one $j \leq m - 1$ satisfying $s_j \geq m - j$ and $\sigma\sigma'(C_0)$ is therefore $(j, m-j)$-full. Let γ_1 be the shortest prefix of $\sigma\sigma'$ so that there is such a value, which we label j_1, satisfying $\gamma_1(C_0)$ is a $(j_1, m - j_1)$-full configuration. Configuration $\gamma_1(C_0)$ must also be m-constrained, because otherwise, there is some index i such that i registers are covered by at least $m - i + 1$ processes in configuration $\gamma_1(C_0)$. But then there is a proper prefix, α, of γ_1 such that $\alpha(C_0)$ is a $(i, m - i)$-full configuration, for some i. Define $C_1 = \gamma_1(C_0)$, $\ell_1 = m$ and $R_1 = \mathcal{R}_{j_1, \ell_1 - j_1}(C_1)$.

In execution $(C_0; \gamma_1)$, each process p in $\mathrm{participants}(\gamma_1)$ leaves the set $\mathrm{idle}(C_0)$ and performs a p-only execution until it is paused when it covers a register. Therefore, $|\mathrm{poised}(C_1, \mathcal{R})| + |\mathrm{idle}(C_1)| = n$. At most $\sum_{c=1}^{j_1}(m - c - 1) + 1$ processes cover registers in R_1. The remainder of at least $n - (\sum_{c=1}^{j_1}(m - c - 1) + 1) > \sum_{c=j_1+1}^{m}(m - c)$ processes are either still idle or are covering registers in $\overline{R_1}$. So for $i = 1$, the following *construction invariant* holds:

(a) $C_i = \gamma_i(C_{i-1})$
(b) $R_{i-1} \subsetneq R_i$
(c) $|\mathrm{poised}(C_i, \overline{R_i})| + |\mathrm{idle}(C_i)| - 1 \geq \sum_{c=j_i+1}^{m}(m - c)$
(d) $j_i \geq j_{i-1} + 1$ and $\ell_i \in \{\ell_{i-1}, \ell_{i-1} - 1\}$ and $\ell_i \leq m$
(e) C_i is a $(j_i, \ell_i - j_i)$-full configuration with $R_i = \mathcal{R}_{j_i, \ell_i - j_i}(C_i)$.

Now suppose that a sequence of tuples $(\gamma_1, C_1, j_1, \ell_1, R_1), \dots, (\gamma_k, C_k, j_k, \ell_k, R_k)$ has been built so that the construction invariant holds for each. If $\ell_k - j_k \geq 3$ and $|\mathrm{idle}(C_k)| \geq 2$ then let B_0, B_1, B_2 to be disjoint sets of processes, such that each covers R_k and each has size $|R_k|$, and let $U = \mathrm{idle}(C_k)$. According to Lemma 4.1, there is a schedule $\beta\sigma\beta'\sigma'$ satisfying:

— β and β' are block writes by B_0 and B_1,
— σ and σ' are concatenations of solo schedules by distinct processes in $\mathrm{idle}(C_k)$,
— $|\mathrm{participants}(\sigma)| \geq \lfloor|\mathrm{idle}(C_k)|/2\rfloor$
— $|\mathrm{participants}(\sigma)| + |\mathrm{participants}(\sigma')| = |\mathrm{idle}(C_k)| - 1$, and
— in configuration $\beta\sigma\beta'\sigma'(C_k)$ all processes in $\mathrm{participants}(\sigma)$ and $\mathrm{participants}(\sigma')$ cover a register in $\overline{R_k}$.

So, combining with (c) of the construction invariant, $|\mathrm{poised}(\beta\sigma\beta'\sigma'(C_k), \overline{R_k})| \geq \sum_{c=j_k+1}^{m}(m - c)$. Hence, there must be a non-empty set of registers $Q' \subseteq \overline{R_k}$ such that each is covered by at least $\ell_k - j_k - |Q'|$ processes. Let γ_{k+1} be the shortest prefix of $\beta\sigma\beta'\sigma'$ such that there is such a Q', which we call Q, in $\gamma_{k+1}(C_k)$ and let $\nu_k = |Q|$, where $\nu_k \geq 1$. Define $C_{k+1} = \gamma_{k+1}(C_k)$, $R_{k+1} = Q \cup R_k$, and $j_{k+1} = \nu_k + j_k$. Then $|R_{k+1}| = \nu_k + j_k = j_{k+1}$. During execution $(C_k; \gamma_{k+1})$, processes that leave $\mathrm{idle}(C_k)$ pause when they cover a register in $\overline{R_k}$. So $|\mathrm{poised}(C_k, \overline{R_k})| + |\mathrm{idle}(C_k)| = |\mathrm{poised}(\gamma_{k+1}(C_k), \overline{R_k})| + |\mathrm{idle}(\gamma_{k+1}(C_k))|$. Therefore, by (c), $|\mathrm{poised}(C_{k+1}, \overline{R_k})| + |\mathrm{idle}(C_{k+1})| - 1 \geq \sum_{c=j_k+1}^{m}(m - c)$.

Furthermore, since γ_{k+1} is chosen to be as short as possible,

$$|\text{poised}(C_{k+1}, Q)| \leq \sum_{c=j_k+1}^{j_k+\nu_k} (\ell_k - c - 1) + 1 < \sum_{c=j_k+1}^{j_k+\nu_k} (m - c)$$

Therefore,

$$|\text{poised}(C_{k+1}, \overline{R_k} \setminus Q)| + |\text{idle}(C_{k+1})| - 1 > \sum_{c=j_k+1}^{m} (m-c) - \sum_{c=j_k+1}^{j_k+\nu_k} (m-c) = \sum_{c=j_{k+1}+1}^{m} (m-c)$$

Thus (a), (b), and (c) of the construction invariant hold for $k+1$. For parts (d) and (e) there are two cases.

Case 1: γ_{k+1} is a prefix of $\beta\sigma$ or $\nu_k \geq 2$. If γ_{k+1} is a prefix of $\beta\sigma$ then there is only one block write to R_k. So in C_{k+1}, each of the j_k registers in R_k remains covered by at least $\ell_k - j_k - 1$ processes and each of the ν_k registers in Q is covered by at least $\ell_k - j_k - \nu_k \leq \ell_k - j_k - 1$ processes. If $\nu_k \geq 2$, then in C_{k+1}, each of the j_k registers in R_k remains covered by at least $\ell_k - j_k - 2$ processes and each of the ν_k registers in Q is covered by at least $\ell_k - j_k - \nu_k \leq \ell_k - j_k - 2$ processes. So in either situation, each of the $j_{k+1} = j_k + \nu_k$ registers in R_{k+1} is covered by at least $\ell_k - j_k - \nu_k = \ell_k - j_{k+1}$ processes. Therefore, setting $\ell_{k+1} = \ell_k$ we have that C_{k+1} is a $(j_{k+1}, \ell_{k+1} - j_{k+1})$-full configuration and the construction invariant holds.

Case 2: $\nu_k = 1$ and γ_{k+1} is not a prefix of $\beta\sigma$. In this case there are two block writes to R_k. So in $\gamma_{k+1}(C_k)$, each register in R_k remains covered by only $\ell_k - j_k - 2$ processes, which is one fewer than the number of processes covering the single register in Q. Since $j_{k+1} = j_k + 1$, we can set $\ell_{k+1} = \ell_k - 1$ to ensure that C_{k+1} is a $(j_{k+1}, \ell_{k+1} - j_{k+1})$-full configuration. So, again the construction invariant holds.

This construction ends in a configuration C_{last} where either $\ell_{\text{last}} - j_{\text{last}} \leq 2$ or $|\text{idle}(C_{\text{last}})| = 1$, since in either case Lemma 4.1 can no longer be applied. Clearly, $C_{\text{last}} = \gamma_1, \gamma_2, \ldots, \gamma_{\text{last}}(C_0)$ so C_{last} is reachable. Since, in $C_{\text{last}-1}$, every register in $R_{\text{last}-1}$ was covered by at least 3 processes, every process in R_{last} is covered by at least one process. So it only remains to bound $|R_{\text{last}}| = j_{\text{last}}$ from below.

First we show that $|\text{idle}(C_{\text{last}})| \leq 1$ is not possible. Intuitively, this is because during execution $(C_0; \gamma_1, \gamma_2, \ldots, \gamma_{\text{last}})$ processes pause in such a way that each of the constructed configurations $C_1, \ldots, C_{\text{last}}$ is m-constrained, which does not allow enough room to use $n-1$ processes.

To make this precise, let γ denote $\gamma_1\gamma_2 \ldots \gamma_{\text{last}}$, and say that process p *is associated with register* r if r is the last register that p covers during execution $(C_0; \gamma)$. During the execution $(C_0; \gamma)$, processes no longer become associated with a register r after r becomes a member of R_i for some i. Let $f(r)$ be the smallest step number, i, such that $r \in R_i$ (and $f(r) = \text{last}$ otherwise). Also, for each register r, let $g(r) = |\{p \mid p \text{ is associated with } r\}|$. We must have $g(r) = \text{poised}(C_{f(r)}, \{r\})$. If $|\text{idle}(C_{\text{last}})| \leq 1$, then each of $n-1$ processes is associated with a register. So $n - 1 \leq \sum_{\mathcal{R}} g(r) = \sum_{\mathcal{R}} \text{poised}(C_{f(r)}, \{r\})$. But by construction, each C_i is ℓ_i-constrained and therefore m-constrained. Thus $\sum_{\mathcal{R}} \text{poised}(C_{f(r)}, \{r\}) \leq \sum_{c=1}^{m}(m-c)$. But then $n - 1 \leq \sum_{c=1}^{m}(m-c) = m(m-1)/2$, which can hold only if $m > \sqrt{2n}$ (since $n \geq 3$).

We can therefore conclude that the construction must have ended because $\ell_{\text{last}} - j_{\text{last}} \leq 2$. So, now we show that if $\ell_{\text{last}} - j_{\text{last}} \leq 2$ then j_{last} is at least $m - \log n - 2$. Let δ be the number of times that Case 2 occurred in the creation of $(\gamma_1, C_1, j_1, \ell_1, R_1), \ldots, (\gamma_{\text{last}}, C_{\text{last}}, j_{\text{last}}, \ell_{\text{last}}, R_{\text{last}})$. Because $\ell_0 = m$ and ℓ_i decreases only for this case and only by one each time, $\ell_{\text{last}} = m - \delta$. Consider a step i where Case 2 occurs, with $\gamma_i = \beta\sigma\beta'\sigma'$. By Lemma 4.1, $|\text{participants}(\sigma)| \geq \lfloor |\text{idle}(C_k)|/2 \rfloor$

so $|\text{participants}(\sigma\sigma')| \geq \lceil|\text{idle}(C_k)|/2\rceil$. Since $\text{idle}(C_0) = n$ and $\text{idle}(C_i) < \text{idle}(C_{i+1})$ it follows that Case 2 can occur at most $\log n$ times. Consequently, $\delta \leq \log n$ implying $\ell_{\text{last}} \geq m - \log n$. Hence, $j_{\text{last}} \geq \ell_{\text{last}} - 2 \geq m - \log n - 2$.

This completes the proof of Theorem 1.2.

5. A SIMPLE ONE-SHOT TIMESTAMPS IMPLEMENTATION USING $\lceil n/2 \rceil$ REGISTERS

Algorithms 1 and 2 implement one-shot timestamps for n processes using $\lceil n/2 \rceil$ registers and thus beat the space used by any register implementation of long-lived timestamps. It is of interest only because of its simplicity; in Section 6, we improve on this space complexity with a more complicated algorithm, which shows that the space lower bound of Section 4 is asymptotically tight.

The `simple-getTS()` method by process p reads each of the registers in sequence, updates the value of the register that p shares by adding one to what p read, and returns as p's timestamp the sum of all the values read. The `simple-compare`(t_1, t_2) method returns the truth value of $t_1 < t_2$.

ALGORITHM 1: simple-compare(t_1, t_2)

```
return t1 < t2;
```

ALGORITHM 2: simple-getTS()

```
// R[1...⌈n/2⌉] is a shared array of multi-reader/2-writer registers each with a
   value in {0,1,2} and initialized to 0. Register R[i] is written by processes 2i
   and 2i+1.
// sum is a local variable
;
sum := 0 ;
for i = 1...⌈n/2⌉ do
    if i = ⌈p/2⌉ then
        R[i] := R[i] + 1;
    end
    sum := sum + R[i];
end
return sum;
```

LEMMA 5.1. *Algorithms 1 and 2 constitute a waitfree implementation of one-shot timestamps for an asynchronous system of n processes.*

PROOF. Clearly both methods `simple-compare` and `simple-getTS` are waitfree. Let p and q be two processors that perform a `simple-getTS` method call and let t_p and t_q be their corresponding timestamps. Assume that p.`simple-getTS()` happens before q.`simple-getTS()`. Each process writes either 1 or 2 to its register and only writes 2 if it observed that its register already held 1. Because it is one-shot, any such observed 1, must have been written by the observing process' partner, and thus the value in each register never decreases. Consequently, the value of sum also never decreases so $t_p \leq t_q$. Since p.`simple-getTS()` happens before q.`simple-getTS`, q's sum will also account for the additional 1 that q writes to its own register and that is not observed by p. Therefore $t_p < t_q$. □

6. AN ASYMPTOTICALLY TIGHT SPACE UPPER BOUND FOR ONE-SHOT TIMESTAMPS

We now present a waitfree algorithm for any timestamp system that invokes at most M `getTS` method calls, which uses $\lceil 2\sqrt{M} \rceil$ registers. In particular, the algorithm uses $\lceil 2\sqrt{n} \rceil$ registers for an n-process one-shot timestamp system, thus establishing Theorem 1.3 and showing that the space lower bound of Section 4 is asymptotically tight.

Timestamps are ordered pairs $(rnd, turn) \in \mathbb{N} \times (\mathbb{N} \cup \{0\})$. The compare method simply compares timestamps lexicographically without accessing shared memory (see Algorithm 3).

ALGORITHM 3: compare$((rnd_1, turn_1), (rnd_2, turn_2))$

return $(rnd_1 < rnd_2) \vee \Big((rnd_1 = rnd_2) \wedge (turn_1 < turn_2)\Big)$

6.1. The getTS algorithm

Algorithm 4 provides the `getTS` method. It uses the parameter m, the number of shared registers, which is a function $m = f(M)$, where M is the maximum number of `getTS` method calls. We will prove that $f(M) = \lceil 2\sqrt{M} \rceil$ suffices. Each process numbers its own `getTS()` method calls sequentially. The k-th time that p invokes `getTS`, it does so using ID $= p.k$. We refer to these IDs as ***getTS-ids***. When specialized to one-shot timestamps, ID can be just the invoking process' identifier.

ALGORITHM 4: getTS(ID)

/* For the k-th invocation by process p, ID $= p.k$. */
Shared:
  $R[1 \ldots m]$: array of multi-writer multi-reader registers, initialized to $\perp$;
Local:
  $r[1 \ldots m]$ initialized to $\perp$;
  j initialized to 1;
  $myrnd$;
while $R[j] \neq \perp$ **do**
  $r[j] = R[j]$
  $j = j + 1$
end
$myrnd = j - 1$
for $j = 1 \ldots myrnd - 1$ **do**
  if $R[myrnd + 1] == \perp$ **then**
    if $r[myrnd].seq[j] == last(R[j].seq)$ **then**
      $R[j] = \langle (\mathrm{ID}), myrnd \rangle$;
      return $(myrnd, j)$
    else if $R[j].rnd < myrnd$ **then**
      $R[j] = \langle (\mathrm{ID}), myrnd \rangle$;
    end
  else
    return $(myrnd + 1, 0)$
  end
end
$r[1 \ldots m] =$ `scan`(*R[1],...,R[m]*)
if $r[myrnd + 1] == \perp$ **then**
  $R[myrnd + 1] = \langle (last(r[1].seq), \ldots, last(r[myrnd].seq), \mathrm{ID}), myrnd + 1 \rangle$
end
return $(myrnd + 1, 0)$

The shared data structure used in the `getTS()` method is an array of m multi-writer multi-reader atomic registers. The content of each register is either $\perp$ (the initial value) or an ordered pair $\langle seq, rnd \rangle$ where, seq is a sequence of getTS-ids, and rnd is a positive integer. The algorithm maintains the invariant that for some integer $k \geq 0$ the first k registers are non-$\perp$ and all other registers are $\perp$. Moreover, the sequence $R[j].seq$ for $j \leq k$ has length either 1 or j. The i-th element of seq is denoted $seq[i]$, and $last(R[j].seq)$ is the last element of the sequence $R[j].seq$.

The algorithm uses the well-known obstruction-free `scan` method due to Afek, Attiya, Dolev, Gafni, Merritt and Shavit [Afek et al. 1993], which returns a *successful-double-collect*. A *collect* reads each $R[1], \ldots, R[m]$ successively and returns the resulting *view*. A successful-double-collect($R[1], \ldots, R[m]$) repeatedly collects until two contiguous views are identical. The `scan` can be linearized at any point between its last two collects. Although this `scan` is not wait-free in general, the use of it by Algorithm 4 is. This is because, in any execution, each `getTS()` performs at most $m-1$ writes, so each `scan` operation will be successful after a finite number of collects. Since `scan` is linearizable, we treat it as atomic for the remainder of this section.

The idea of the algorithm is as follows. An execution proceeds in phases. During phase k, $R[1]$ through $R[k-1]$ are non-$\perp$; $R[k+1]$ to $R[m]$ are $\perp$; $R[k]$ is either written or some `getTS()` is poised to write to it for the first time. Every write to $R[k]$ during phase k is a pair $\langle seq, rnd \rangle$, which stores a sequence of k getTS-ids in seq. We say that register $R[i]$ is *valid* if the phase is k and the last entry stored in $R[i].seq$ equals the i-th entry stored in $R[k].seq$.

Roughly speaking, phase $k-1$ ends when some `getTS(`q`)` method discovers that all registers $R[1]$ through $R[k-1]$ are invalid. Then `getTS(`q`)` performs a `scan`, which returns the view $(r_1, \ldots, r_{k-1}, \perp, \ldots, \perp)$. The k-th phase starts precisely at this scan. Then `getTS(`q`)` prepares to write the sequence $(\ell_1, \ldots, \ell_{k-1})$ to $R[k].seq$, where $\ell_i = last(r_i.seq)$ for $1 \leq i \leq k-1$. First imagine, for simplicity, that `getTS(`q`)`'s scan and subsequent write to $R[k]$ occur in one atomic operation. In that case, at the beginning of the k-th phase, every register $R[i]$, $1 \leq i \leq k-1$, is valid. Because `getTS(`q`)` started phase k, it returns the timestamp $(k, 0)$.

For the rest of phase k, any other `getTS(`p`)` method that began in phase k examines the registers $R[1]$ through $R[k-1]$ in this order looking for the first register that is valid. If it finds one, say $R[i]$, it writes $\langle p, k \rangle$ to $R[i]$, thus *invalidating* $R[i]$ by making $last(R[i].seq)$ differ from the i-th entry stored in $R[k].seq$, and returns the timestamp (k, i). If it fails to find one, it will perform a scan and prepare to start phase $k+1$. Observe that this algorithm works correctly if all `getTS()` calls are sequential: the `getTS()` that starts phase k returns $(k, 0)$ and the j-th `getTS()` call after that, for $1 \leq j \leq k-1$, invalidates $R[j]$ and returns (k, j).

There are several complications and subtleties that arise due to concurrent `getTS()` executions. Suppose a `getTS()` that began in phase k sleeps before it writes its invalidation to a register $R[i]$. If it wakes up during some later phase k', its write could invalidate $R[i]$ for phase k' making timestamp (k', i) unusable, and so increase the space requirements. Such damage is confined to at most one such wasted timestamp per `getTS()` method as follows. Each `getTS(`p`)` begins by setting its local variable, $myrnd_p$, to the largest value such that $R[myrnd_p]$ is non-$\perp$. Before each of its writes, `getTS(`p`)` checks that $R[myrnd_p + 1]$ is still non-$\perp$. If it is not, the phase must have advanced, and `getTS(`p`)` can safely terminate with timestamp $(myrnd_p + 1, 0)$.

A more serious potential problem due to concurrency occurs when our simplifying assumption above (that the scan and subsequent write occur in one atomic operation) does not hold. Suppose at the end of phase $k-1$, both `getTS(`p`)` and `getTS(`q`)` are poised to execute a `scan` and then write the result into $R[k]$. If, after `getTS(`p`)`'s scan and before `getTS(`q`)`'s scan, some "old" writes happen to some registers say $R[1], \ldots, R[j]$, the results of their `scan`s will differ. After both `scan`s, `getTS(`q`)`'s view matches all register values, but `getTS(`p`)`'s view matches only the contents of $R[j+1]$,..., $R[k-1]$. Now let both `getTS(`p`)` and `getTS(`q`)` proceed until both are poised to write the result computed from their view to $R[k]$, and suppose `getTS(`p`)` writes first. At this point, registers $R[1], \ldots, R[j]$ are already invalid because of `getTS(`p`)`'s out-of-date view. Another `getTS(`a`)` starting at this point will invalidate $R[j+1]$ and return timestamp $(k, j+1)$. If after that, `getTS(`q`)` writes to $R[k]$, the first j registers could become valid,

and getTS(b) beginning after getTS(a) completes would invalidate $R[1]$ and return timestamp $(k, 1)$, which is incorrect because it is less than getTS(a)'s timestamp. This problem is eliminated by ensuring that when getTS(a) determines that a register $R[i]$ is invalid, it will remain invalid for the duration of the phase. One way to achieve this is to have getTS(a) overwrite the invalid register $R[i]$ with $\langle a, myrnd_a\rangle$ before it moves on to investigates the validity of $R[i+1]$. This simple repair to correctness, however, can increase space complexity. Instead, the overwriting by getTS(a) is done only when necessary. Specifically, getTS(a) determines that a register $R[i]$ is invalid by reading a pair $\langle seq_i, rnd_i\rangle$ from $R[i]$ and finding that $last(seq_i)$ is not equal to its view of the i-th value in $R[k].seq$. If $rnd_i = k$ then this invalidation cannot be due to an old write from an earlier phase, so no overwriting is needed. In the algorithm, getTS(a) overwrites register $R[i]$ with $\langle a, k\rangle$ only when it read $rnd_i < k$.

As we shall see, these additional techniques are enough to convert the idea of a timestamp object that is correct under sequential accesses to an algorithm for concurrent timestamps that is correct (Lemma 6.4) and space efficient (Lemma 6.5) and waitfree (Lemma 6.14). These three lemmas, when specialized to the one-shot case, constitute the proof of Theorem 1.3.

6.2. Algorithms 3 and 4 Correctly Implement Timestamps

We isolate some properties of Algorithm 4 that will serve to simplify both the correctness and complexity arguments. In the following, the local variable x in the code of Algorithm 4 is denoted by x_{id} when it is used in the method call of getTS(id).

CLAIM 6.1. *In any execution*

(a) once the content of a shared register becomes non-$\perp$ it remains non-$\perp$; and
(b) For any j, $1 \le j \le m$, the value of last($R[j].seq$) that is written by each write to $R[j]$ is distinct.

In any configuration of an execution

(c) if any getTS(id) *has returned* $(rnd, turn)$ *then* $R[rnd] \neq \perp$*; and*
(d) if $R[k] \neq \perp$ then $\forall k' \le k$, $R[k'] \neq \perp$.

PROOF.

(a) No getTS() method call ever writes $\perp$ to any shared register.
(b) The only writes to a shared register occur at lines 8, 11 and 15. In any single instance of getTS, say getTS(id), in each iteration j of the for-loop (line 5), for $1 \le j \le myrnd_{id} - 1$, at most one write occurs, either at line 8 or 11 but not both, and any such write is to $R[j]$. If getTS(id) writes at line 15, it writes to $R[myrnd_{id} + 1]$. So, in any single execution of getTS(id), each register is written at most once. Every write by getTS(id) to a register $R[j]$ sets $last(R[j].seq)$ to id, which is distinct for each getTS method call.
(c) getTS(id) returned at line 9, 12 or 16. We show that in all cases the register $R[rnd]$ was written before getTS(id) returned. Then the claim follows by (a). If getTS(id) returned in line 9 then $rnd = myrnd_{id}$, and $R[myrnd_{id}] \neq \perp$ when the while-loop of getTS(id) completes. If getTS(id) returned in line 12, then $rnd = myrnd_{id} + 1$. Before returning, however, getTS(id) evaluated the if-statement in line 6 to be false, implying $R[myrnd_{id} + 1] \neq \perp$. If getTS($id$) returned in line 16, then $rnd = myrnd_{id} + 1$ and either getTS(id) evaluated the if-statement in line 14 to be false, or getTS(id) wrote to $R[myrnd_{id} + 1]$ in line 15. In either case, $R[myrnd_{id} + 1] \neq \perp$ before getTS(id) returned.
(d) Consider any write to a register $R[k]$ and suppose it occurs in the execution of getTS(id). The while-loop of getTS(id) confirms that all registers $R[1]$ through

$R[myrnd_{id}]$ were previously non-$\perp$, before any write by `getTS(id)`. Writes only occur in lines 8, 11 and 15 of `getTS`. Every write by `getTS(id)` in lines 8 and 11 is to some register $R[j]$ where $j < myrnd_{id}$. A write in line 15 by `getTS(id)` is to $R[myrnd_{id} + 1]$. So in all cases, when the write to $R[k]$ occurred, registers $R[1]$ through $R[k-1]$ were previously non-$\perp$. The claim follows by (a).

□

Definition 6.2. A `getTS()` method *fails in iteration j in line 6* if, in its j-th iteration of the for-loop (line 5), the if-condition in line 6 returns false; it *fails in iteration j in line 7* if, in its j-th iteration of the for-loop, the if-condition in line 7 returns false; and it *fails in iteration j*, if either it fails in iteration j in line 6 or it fails in iteration j in line 7.

CLAIM 6.3. *If $myrnd_p \geq myrnd_q$ for two method calls* `getTS(p)` *and* `getTS(q)`*, and* `getTS(p)` *writes to $R[j]$ before the j-th iteration of the for-Loop of* `getTS(q)` *begins, then* `getTS(q)` *fails in iteration j.*

PROOF. $R[myrnd_p] \neq \perp$ when `getTS(p)` executed line 1 of its while-loop for $j = myrnd_p$, and thus by Claim 6.1(a) remains non-$\perp$ after the while-loop completes.

First, suppose $myrnd_p > myrnd_q$. By Claim 6.1(a) and (d), $R[myrnd_q + 1] \neq \perp$ when `getTS(q)` executes its j-th iteration of the for-loop. So the if-condition in line 6 of `getTS(q)` returns false, and `getTS(q)` fails at iteration j.

Now, suppose $myrnd_p = myrnd_q$. `getTS(p)` wrote to $R[j]$ after executing its while-loop and therefore after $R[myrnd_q]$ became non-$\perp$. The content of $r[myrnd_q]_q$, which q read from $R[myrnd_q]$, came from the value of a scan taken during the execution of some `getTS` when $R[myrnd_q] = \perp$. Hence when `getTS(q)` executes line 7 of the j-th iteration of the for-loop, it compares the value of $r[myrnd_q]_q.seq[j]$, which is the value of $last(R[j].seq)$ that $R[j]$ had before $R[j]$ was written by `getTS(p)`, to a value of $last(R[j].seq)$ that $R[j]$ had after this write. So by Claim 6.1(b), this comparison must return false, and `getTS(q)` fails at iteration j. □

LEMMA 6.4. *Provided $m = f(M)$ is large enough, Algorithms 3 and 4 implement a timestamp object for any asynchronous shared memory system of processes that invokes* `getTS()` *a total of at most M times.*

PROOF. Let `getTS(p)` and `getTS(q)` return timestamps $(rnd_p, turn_p)$ and $(rnd_q, turn_q)$ respectively. We need to show that if `getTS(p)` happens before `getTS(q)`, then compare$((rnd_p, turn_p), (rnd_q, turn_q))$ returns true. That is, we need to show that $(rnd_p < rnd_q) \vee \big((rnd_p = rnd_q) \wedge (turn_p < turn_q)\big)$.

By Claim 6.1(c), after `getTS(p)` completes, $R[rnd_p] \neq \perp$. Therefore by Claim 6.1(a) and (d), $R[1], \ldots, R[rnd_p] \neq \perp$ throughout the method call `getTS(q)`. Thus, at line 4 after the while-loop, `getTS(q)` sets $myrnd_q \geq rnd_p$. If `getTS(q)` returns at line 12 or 16, $rnd_q = myrnd_q + 1 \geq rnd_p + 1$ implying $(rnd_p < rnd_q)$ so compare$((rnd_p, turn_p), (rnd_q, turn_q)$ returns true as required. If `getTS(q)` returns at line 9, and $myrnd_q > rnd_p$, then again compare$((rnd_p, turn_p), (rnd_q, turn_q)$ returns true. Therefore, suppose that `getTS(q)` returns at line 9 and $rnd_q = myrnd_q = rnd_p$. In this case, $turn_q$ is non-zero. If p returns at line 12 or 16, $turn_p$ is zero. So again compare$((rnd_p, turn_p), (rnd_q, turn_q)$ returns true.

The only remaining case is that both `getTS(p)` and `getTS(q)` return at line 9 and $rnd_q = myrnd_q = rnd_p = myrnd_p$. In this case, we show that $turn_q > turn_p$ by proving that `getTS(q)` fails at every iteration 1 through $turn_p$. By Lemma 6.3, it suffices to show that for every j, $1 \leq j \leq turn_p$, there is some `getTS(p')`, with $myrnd_{p'} \geq myrnd_q$ that writes to $R[j]$ before `getTS(q)` begins iteration j. For `getTS(p)`, the if-condition in

line 7 must have returned false for all iterations 1 through $turn_p - 1$, and then returned true in iterations $turn_p$. For $j < turn_p$, when `getTS(p)` fails at iteration j, it reads $R[j]$ (line 10). If this read shows $R[j].rnd \geq myrnd_p$ there must be a `getTS(p')`, with $myrnd'_p \geq myrnd_p$ that wrote this. If the read shows $R[j].rnd < myrnd_p$, then `getTS(p)` itself writes to $R[j]$ changing $R[j].rnd$ to $myrnd_p$ (line 11). For $j = turn_p$ process p itself writes into register $R[j]$. In all cases, the write happened before `getTS(q)`. □

6.3. Space Complexity of Algorithm 4

Fix an arbitrary execution E that contains at most M `getTS()` invocations. In this subsection we prove no register beyond $R[\lceil 2\sqrt{M} \rceil]$ is accessed in E.

The proof proceeds as follows. We partition E into *phases*. Phase 0 starts at the beginning of E. Phase $\varphi \geq 1$ starts at the point of the first `scan` (line 13) by some `getTS(p)`, for which $myrnd_p = \varphi - 1$. We say that phase φ *completes during* E, if phase $\varphi + 1$ starts during E. Call the first write to $R[j]$ during any phase an *invalidation write*. First, Claim 6.8 shows that only registers $R[1]$ through $R[\varphi]$ can be written during phase φ. Next, Claim 6.10 establishes that if phase φ completes then it contains exactly φ invalidation writes. Finally, we define a charging mechanism that charges each invalidation write in E to some write in E in such a way that there are at most 2 charges to all the writes of any one `getTS()` invocation. This gives us Claim 6.13, which states that there are a total of at most $2M$ invalidation writes.

Therefore, the total number of phases, Φ, satisfies: $\sum_{\varphi=1}^{\Phi} \varphi \leq 2M$. Hence, $\Phi < 2 \cdot \sqrt{M}$. The algorithm uses a final sentinel register that is read but never written, and always contains the initial value $\perp$. So at most $\lceil 2 \cdot \sqrt{M} \rceil$ registers are accessed in E. Therefore, once Claims 6.8, 6.10 and 6.13 are proved (below) we have the following:

LEMMA 6.5. *Algorithm 4 uses at most $\lceil 2\sqrt{M} \rceil$ registers for M `getTS()` operations.*

Technical claims

The proof of Lemma 6.5, via Claims 6.8, 6.10 and 6.13, is the most challenging part of our arguments concerning Algorithm 4. First, we encapsulate the relationship between the value of the variable $myrnd_p$ of a `getTS(p)` method call and the phase number φ during which `getTS(p)` writes to certain registers.

CLAIM 6.6.

(a) If `getTS(p)` writes to register $R[i]$ when $R[i+1] = \perp$, then that write occurs in line 15.
(b) `getTS(p)` executes line 15 during some phase $\varphi \geq myrnd_p + 1$.
(c) `getTS(p)` executes line 4 during some phase $\varphi' \geq myrnd_p$.

PROOF. (a) If w is a write by `getTS(p)` to $R[i]$ in line 8 or 11, then $i \leq myrnd_p - 1$. When `getTS(p)` previously read $R[myrnd_p]$ in line 2, its value was non-$\perp$, so, by Claim 6.1(a) and (d) $R[i+1]$ is non-$\perp$ when w occurred. Hence, any write to $R[i]$ when $R[i+1] = \perp$ could not have occurred at line line 8 or 11, and thus could only occur at line 15.

(b) When `getTS(p)` executes line 15, it has already executed its `scan` in line 13. By definition of phase, if phase $myrnd_p+1$ had not already begun before this `scan` occurred, then it began with this scan.

(c) Before `getTS(p)` executes line 4, its while-loop terminated because $R[myrnd_p] \neq \perp$ and $R[myrnd_p + 1] = \perp$. By (a), $R[myrnd_p]$ must have previously changed from $\perp$ to non-$\perp$, when some `getTS(r)` executed line 15. When `getTS(r)` executes this write, it wrote to $R[myrnd_r + 1]$, so $myrnd_r + 1 = myrnd_p$. Before this write, `getTS(r)` executed a

`scan` at line 13, which either started phase $myrnd_r+1$, or phase $myrnd_r+1$ had already started. Thus, $myrnd_r + 1 = myrnd_p$ started before `getTS(p)` executed line 4. □

CLAIM 6.7. *If during phase $myrnd_p$, `getTS(p)` fails iteration i at line 7, then during phase $myrnd_p$ and before the failure, there was a write to $R[i]$ and a write to $R[myrnd_p]$.*

PROOF. Let $v = (v_1, \ldots, v_k)$ be the value of the sequence stored in $R[myrnd_p].seq$ when `getTS(p)` reads that register in line 2. Let `getTS(b)` be the method call that wrote v to $R[myrnd_p].seq$. The while-loop of `getTS(p)` terminated when `getTS(p)` read $R[myrnd_p + 1] = \perp$ in the $(myrnd_p + 1)$-th iteration of its while-loop after reading $R[myrnd_p] \neq \perp$ in its previous iteration. By Claim 6.1 (a), $R[myrnd_p + 1] = \perp$ when `getTS(b)` wrote v to $R[myrnd_p]$. Therefore, by Claim 6.6 (a), `getTS(b)` wrote to $R[myrnd_p]$ in line 15 and thus $myrnd_b = myrnd_p - 1$. By Claim 6.6 (b), `getTS(b)`'s write to $R[myrnd_p]$ happens during phase $myrnd_p$ or a later phase. Because this write happens before `getTS(p)` reads $R[i]$ in line 7 during phase $myrnd_p$, `getTS(b)`'s write to $R[myrnd_p]$ occurs during phase $myrnd_p$ before `getTS(p)` fails at iteration i.

Before executing line 15, `getTS(b)` executed a `scan` in line 13, and obtained v_i for the value of $last(R[i].seq)$. By the assumption that `getTS(p)` fails at iteration i in line 7, $v_i \neq last(R[i].seq)$ when `getTS(p)` reads $R[i]$ in line 7. Therefore, there must have been a write to $R[i]$ between `getTS(b)`'s scan and `getTS(p)`'s read at line 7. This write must have occurred in phase $myrnd_p$ because, by the definition of phase, either phase $myrnd_p$ began before `getTS(b)`'s scan or it began at this scan. Thus, a write to $R[i]$ occurs in phase $myrnd_p$ before `getTS(p)` fails iteration i. □

Counting invalidation writes per completed phase

Our goal is to show that during phase φ there is exactly one invalidation write to each register $R[1]$ through $R[\varphi]$, and no other registers are written.

CLAIM 6.8. *Only registers $R[1], \ldots, R[\varphi]$ are written during phase φ.*

PROOF. Until some `getTS()` has executed line 15, and thus phase 1 has started, no register is written. Hence, the claim is trivially true for $\varphi = 0$. Now let $\varphi \geq 1$. If `getTS(q)` writes to $R[j]$ in lines 8 or 11, then by Claim 6.6 (c), $\varphi \geq myrnd_q$, and by the semantics of the for-loop, $j < myrnd_q$. If q writes to $R[j]$ in line 15, then $j = myrnd_q + 1$ and by Claim 6.6 (b), $\varphi \geq myrnd_q + 1$. Hence, in either case $j \leq \varphi$. □

CLAIM 6.9. *If phase φ completes in E, then for each $1 \leq j \leq \varphi$, there is at least one write to $R[j]$ during phase φ.*

PROOF. By definition, phase $\varphi + 1 \geq 1$ begins at the first `scan` (line 13) by some `getTS(p)`, for which $myrnd_p = \varphi$. Since `getTS(p)` executes line 13, its call does not return during the for-loop. Therefore, this scan can happen only if this `getTS(p)` fails in iteration j at line 7 for every $1 \leq j \leq \varphi - 1$. Thus, by Claim 6.7, a write to register $R[\varphi]$ and a write to $R[j]$ for every $1 \leq j \leq \varphi - 1$ happens in phase φ. □

CLAIM 6.10. *There are exactly φ invalidation writes in any completed phase φ.*

PROOF. By Claims 6.8 and 6.9, exactly the registers $R[1]$ through $R[\varphi]$ are written during phase φ. The set of first writes to each of these registers during phase φ is, by definition, the set of invalidation writes in phase φ. □

Counting all invalidation writes

We rely on some definitions and factor out some sub-claims to facilitate the proof of Claim 6.13 below.

CLAIM 6.11. *If a write at line 11 by* `getTS(p)` *happens during phase* $myrnd_p$*, then that write is not an invalidation write.*

PROOF. Let w be a write at line 11 by `getTS(p)` to register $R[j]$ that occurs during phase $myrnd_p$. Then w happens only if `getTS(P)` fails at iteration j at line 7. So, according to Claim 6.7, there was a previous write to $R[j]$ during phase $myrnd_p$. Hence, w is not an invalidation write. □

There can be an interval between when the first `getTS(q)` with $myrnd_q = \varphi - 1$ does a scan at line 13 thus starting phase φ, and when the first write to $R[\varphi]$ happens, which is the first point at which other `getTS()` method calls can discern that the current phase is (at least) φ. To capture this, say that the phase φ is *invisible* from the beginning of phase φ to the step before the first write to $R[\varphi]$ and *visible* from the first write to $R[\varphi]$ to the end of phase φ.

CLAIM 6.12. *Any write by* `getTS(p)` *at line 8 or at line 15 or any write at line 11 that happens after the phase* $myrnd_p + 1$ *becomes visible, is* `getTS(p)`*'s last write.*

PROOF. Algorithm 4 returns after any write at line 8 or line 15, so such a write is the last write of the method call. Now consider a line 11 write w by `getTS(p)`. If w happens anytime after phase $myrnd_p + 1$ becomes visible, then after w, `getTS(p)` will discern that the phase number has increased when it reads $R[myrnd_p + 1]$ to be non-$\perp$ either at line 6 in the next iteration of the for-loop or, if the for-loop is complete, at line 14. In either case `getTS(p)` returns without another write, so such a write is also the last write of the method call. □

CLAIM 6.13. *There are at most* $2M$ *invalidation writes in execution* E*.*

PROOF. Define:

$A = \{w \mid w$ is a first invalidation write by some `getTS()` method$\}$

$B = \{w \mid w$ is the last write by some `getTS()` method and w is an invalidation write$\}$

$C = \{w \mid w$ is the last write by some `getTS()` method and w is not an invalidation write$\}$

Let W^* be the disjoint union of A, B and C. Since each `getTS()` can have at most one write in A and at most one write in either B or C but not both it follows that $|W^*| \leq 2M$. Let W be the set of all writes, and let I be the set of all invalidation writes during execution E. We will define a function $f : I \to W$. Then, it will suffice to show that f is injective with range a subset of W^*.

Define:

$I_1 = \{w \mid w$ is an invalidation write at line 8 or 15$\}$

$I_2 = \{w \mid \exists$ `getTS(p)` satisfying (w is an invalidation write at line 11 by `getTS(p)`) and (w happens after the beginning of the visible part of phase $myrnd_p + 1$) $\}$

$I_3 = \{w \mid \exists$ `getTS(p)` satisfying (w is an invalidation write at line 11 by `getTS(p)`) and (w happens during the invisible part of phase $myrnd_p + 1$) and (w is `getTS(p)`'s first invalidation write)$\}$

$I_4 = \{w \mid \exists$ `getTS(p)` satisfying (w is an invalidation write at line 11 by `getTS(p)`) and (w happens during the invisible part of phase $myrnd_p + 1$) and (w is not `getTS(p)`'s first invalidation write)$\}$

Let w be a write by `getTS(p)`. Then w happens at line 8, or line 11 or line 15, and, by Claim 6.6(c), w happens in some phase φ satisfying $\varphi \geq myrnd_p$. By Claim 6.11, if w is a write at line 11 that occurs during phase $myrnd_p$, then w is not an invalidation

write. Hence $I_1 \cup I_2 \cup I_3 \cup I_4 = I$. Also, clearly I_1, I_2, I_3 and I_4 are mutually disjoint. Therefore $\{I_1, I_2, I_3, I_4\}$ is a partition of I.

For all $w \in I_1 \cup I_2 \cup I_3$ define $f(w) = w$. By definition, $I_3 \subseteq A$, and by Claim 6.12, $I_1 \cup I_2 \subseteq B$. So, f maps $I_1 \cup I_2 \cup I_3$ to the disjoint union of A and B and clearly, f is injective on $I_1 \cup I_2 \cup I_3$.

It remains to map I_4 to C and show this map is injective. Let w be a write in I_4 by `getTS(p)` to register $R[i]$. By definition of I_4, w occurs during the invisible part of phase $myrnd_p + 1$, and there is another invalidation write, say u, by `getTS(p)` that precedes w in E. Claims 6.6 (c), 6.11 and 6.12 imply that u is a line 11 write that also occurs during the invisible part of phase $myrnd_p + 1$.

In line 10, before executing w, `getTS(p)` reads a value $x < myrnd_p$ from $R[i].rnd$. Let o be this read operation. Operation o occurs after u and before w and thus also during the invisible part of phase $myrnd_p + 1$. Define $f(w)$ to be the write operation that wrote the value to $R[i]$ that was read by o.

We now show that $f(w)$ is in C. Let `getTS(a)` be the method call that starts phase $myrnd_p + 1$ by executing a `scan` in line 13. Then $myrnd_a = myrnd_p$, and during phase $myrnd_p$, `getTS(a)` fails at iteration j at line 7 and thus executes line 10, for all $j = 1, \ldots, myrnd_p - 1$. In particular for $j = i$, `getTS(a)` either writes the value $myrnd_a = myrnd_p > x$ to $R[i].rnd$ in line 11, or in line 10 it reads $R[i].rnd \geq myrnd_a > x$. Hence, $f(w)$, which writes the value x to $R[i].rnd$ that is read by o, must happen after the i-th iteration of `getTS(a)`'s for-loop and before o. Furthermore, $f(w)$ cannot happen during phase $myrnd_p + 1$, because otherwise w would not be an invalidation write. We conclude that $f(w)$ is a write to $R[i]$ that happened in phase $myrnd_a$ after `getTS(a)` failed at iteration i, and hence, by Claim 6.7, $f(w)$ is not an invalidation write.

Let `getTS(b)` be the method call that executes $f(w)$. When `getTS(a)` finished its while-loop, $R[myrnd_a] = R[myrnd_p] \neq \perp$. Hence, by Claim 6.1 (a), $R[myrnd_p] \neq \perp$ when $f(w)$ occurs. Since $myrnd_b = x < myrnd_p$, by Claim 6.1 (a) and (d), $R[myrnd_b + 1] \neq \perp$ when `getTS(b)` executes either the if-statement in line 6 (in iteration $i + 1$ of its for-loop) or in line 14 (if its for-loop is completed because $i = myrnd_b - 1$). In either case, `getTS(b)` returns without another write. Therefore, $f(w)$ is the last write by `getTS(b)` and is not an invalidation write, so $f(w) \in C$ by definition of C. Finally, we show that $f(\cdot)$ on the domain I_4 is injective. If $f(w)$ occurs in phase φ then, as we have just seen, w occurs during the invisible part of phase $\varphi + 1$. Suppose $f(w) = f(w')$ where $w, w' \in I_4$. Then w and w' are both invalidation writes to the same register during the same phase $\varphi + 1$. But this is impossible since there can be only one invalidation write per register per phase. $\square$

6.4. Algorithms 3 and 4 are Waitfree

LEMMA 6.14. *Algorithms 3 and 4 are wait-free provided the bound M on the number of allowed* `getTS` *method call is fixed in advance.*

PROOF. Clearly, `compare` is wait-free. The number of registers m provided for `getTS()` is at least one more than the number that can be written, so the last register $R[m]$ is always $\perp$. Since each iteration of the while-loop increments j until $R[j] = \perp$ is read, the while-loop can iterate at most $m-1$ times. Similarly, since $myrnd$ is the index of a non-$\perp$ register, the for-loop can iterate at most $m - 2$ times. All, operations inside and outside the while and for loops, except the `scan`, are wait-free. Hence, it remains to show that all calls of `scan` terminate within a bounded number of steps. It is immediate from the code that each `getTS()` writes to each register at most once, implying each `getTS()` method writes fewer than $m = \lceil 2\sqrt{M} \rceil$ times. Thus, after a finite number of reads during the `scan`, the scanning process must see no more changes to registers, and so will achieve a successful double collect and terminate. $\square$

7. FURTHER REMARKS

The lower and upper bounds for long-lived and one-shot timestamps compare and contrast in several ways. In the execution constructed in the lower bound for one-shot timestamps, each process that participates in a block write, takes no further steps in the computation. As a consequence, the proof actually applies without change if each register is replaced by any historyless object. The asymptotically matching upper bound is, however, achieved using registers. In contrast, our proof of the lower bound for long-lived timestamps does not extend to historyless objects. So it remains an open question whether there is an implementation of long-lived timestamps from a sub-linear number of historyless objects. Both the long-lived and the one-shot lower bounds apply even to non-deterministic solo-terminating algorithms, while the asymptotically matching algorithms are wait-free.

The upper bound for one-shot timestamps applies for any bounded number of `getTS()` method invocations. The covering argument in the proof of the lower bound, however, prevents any similar generalization: it depends on each process invoking at most one `getTS()`.

The one-shot algorithm generalizes even to the situation where the number of `getTS()` method invocations is not bounded, provided that the system could acquire additional registers as needed. In this case however, progress would be non-blocking only instead of wait-free.

ACKNOWLEDGMENTS

The authors thank Faith Ellen for valuable comments on an earlier draft of parts of this paper.

REFERENCES

ABRAHAMSON, K. R. 1988. On achieving consensus using a shared memory. In *PODC*. 291–302.

AFEK, Y., ATTIYA, H., DOLEV, D., GAFNI, E., MERRITT, M., AND SHAVIT, N. 1993. Atomic snapshots of shared memory. *J. ACM 40,* 4, 873–890.

AFEK, Y., DOLEV, D., GAFNI, E., MERRITT, M., AND SHAVIT, N. 1994. A bounded first-in, first-enabled solution to the l-exclusion problem. *ACM Transactions on Programming Languages and Systems 16,* 3, 939–953.

ATTIYA, H. AND FOUREN, A. 2003. Algorithms adapting to point contention. *Journal of the ACM 50,* 4, 444–468.

BURNS, J. E. AND LYNCH, N. A. 1993. Bounds on shared memory for mutual exclusion. *Inf. Comput. 107,* 2, 171–184.

DOLEV, D. AND SHAVIT, N. 1997. Bounded concurrent time-stamping. *SIAM Journal on Computing 26,* 2, 418–455.

DWORK, C. AND WAARTS, O. 1999. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *Journal of the ACM 46,* 5, 633–666.

ELLEN, F., FATOUROU, P., AND RUPPERT, E. 2008. The space complexity of unbounded timestamps. *Distributed Computing 21,* 2, 103–115.

FICH, F. E., HERLIHY, M. P., AND SHAVIT, N. 1998. On the space complexity of randomized synchronization. *Journal of the ACM 45,* 5, 843–862.

FIDGE, C. J. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference (ACSC'88)*. 56–66.

FISCHER, M. J., LYNCH, N. A., BURNS, J. E., AND BORODIN, A. 1989. Distributed fifo allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems 11,* 1, 90–114.

GAWLICK, R., LYNCH, N. A., AND SHAVIT, N. 1992. Concurrent timestamping made simple. In *1st Israel Symposium on Theory of Computing Systems (ISTCS)*. 171–183.

GUERRAOUI, R. AND RUPPERT, E. 2007. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing 20,* 3, 165–177.

HALDAR, S. AND VITÁNYI, P. M. B. 2002. Bounded concurrent timestamp systems using vector clocks. *J. ACM 49,* 1, 101–126.

ISRAELI, A. AND LI, M. 1993. Bounded time-stamps. *Distributed Computing 6,* 4, 205–209.

ISRAELI, A. AND PINHASOV, M. 1992. A concurrent time-stamp scheme which is linear in time and space. In *Distributed Algorithms, 6th International Workshop (WDAG)*. 95–109.

LAMPORT, L. 1974. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM 17,* 8, 453–455.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21,* 7, 558–565.

LI, M., TROMP, J., AND VITÁNYI, P. M. B. 1996. How to share concurrent wait-free variables. *Journal of the ACM 43,* 4, 723–746.

MATTERN, F. 1989. Virtual time and global states of distributed systems. In *Proc. Workshop on Parallel and Distributed Algorithms*. 215–226.

RICART, G. AND AGRAWALA, A. K. 1981. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM 24,* 1, 9–17.

SARIN, S. K. AND LYNCH, N. A. 1987. Discarding obsolete information in a replicated database system. *IEEE Trans. Software Eng. 13,* 1, 39–47.

VITÁNYI, P. M. B. AND AWERBUCH, B. 1986. Atomic shared register access by asynchronous hardware. In *27th Annual Symposium on Foundations of Computer Science (FOCS)*. 233–243.

WUU, G. T. J. AND BERNSTEIN, A. J. 1986. Efficient solutions to the replicated log and dictionary problems. *Operating Systems Review 20,* 1, 57–66.